2015 International Conference on Parallel Architecture and Compilation

An Efficient, Self-Contained, On-Chip Directory: DIR_1-SISD

Mahdad Davari*, Alberto Ros†, Erik Hagersten* and Stefanos Kaxiras*
*Dept. of Information Technology
Uppsala University, Sweden
Emails: {mahdad.davari, erik.hagersten, stefanos.kaxiras}@it.uu.se
†Dept. of Computer Engineering
University of Murcia, Spain
Email: aros@ditec.um.es

***Abstract*—Directory-based cache coherence is the de-facto standard for scalable shared-memory multi/many-cores and significant effort is invested in reducing its overhead. However, directory area and complexity optimizations are often antithetical to each other. Novel directory-less coherence schemes have been introduced to remove the complexity and cost associated with directories in their entirety. However, such schemes introduce new challenges by transferring some of the directory complexity and functionality to the OS and using the page table and the TLBs to store data classification information.**

In this work we bridge the gap between directory-based and directory-less coherence schemes and propose a hybrid scheme called DIR_1-SISD which employs self-invalidation and self-downgrade as directory policies for the *shared* entries. DIR_1-SISD allows simultaneous optimizations in area and complexity without relying on the OS. DIR_1-SISD keeps track of a single —private— owner, *or* allows multiple-readers-multiple-writers to exist simultaneously by transferring the responsibility for their coherence to the corresponding cores. A DIR_1-SISD *self-contained* directory cache has a unique ability to minimize eviction-induced complexities by allowing directory entries to be evicted without maintaining inclusion with the cached data (thus avoiding the complexities of broadcasts) and without the need to have a backing store. Using simulation we show that a small, self-contained, DIR_1-SISD cache outperforms a traditional DIR_n-NB MESI protocol with a directory cache embedded in the LLC (8% in execution time and 15% in traffic) and, further, outperforms a SISD protocol that relies on the OS to provide a persistent page-based directory (4% in execution time and 20% in traffic).

***Keywords*-multicore; memory hierarchy; cache coherence;**

I. Introduction

Hardware-based cache coherence has long served as an enabling factor in harnessing the compute power of multi/many-cores by providing an easy programming paradigm through sparing programmers from dealing with explicit cache and consistency management [1]. While snoop-based coherence schemes allowed implementation of shared-memory systems using conventional bus-based networks [2]–[4], the need for scalable architectures, incorporating ever-increasing number of cores, necessitated directory-based coherence schemes [5]–[10]. Directory-based cache coherence, which has served in many shared-memory chip multi-processor (CMP) designs, has been extensively used and studied [11], [12].

At a higher level of abstraction, different directory schemes are distinguished based on (i) how they keep track of sharers, and (ii) which policy they employ to maintain coherence across those sharers. The notation DIR_i-X has been used to describe directory schemes [11], [12], where index *i* refers to how the sharers are tracked, and *X* denotes the policy to maintain coherence across the sharers, such as *broadcast (B)* or *no-broadcast (NB)*. Directory policy, on the other hand, defines how the directory scheme behaves with respect to (i) maintaining single-writer-{single or multiple}-reader invariant [11], [13], (ii) request forwarding [12], and (iii) directory entry eviction —in case of using a directory cache, which is often the case [8], [9]. Directory complexity is mainly attributed to the complexities associated with the directory policies. As an example, the directory scheme DIR_n-NB is the extreme case which seeks to eliminate the complexities associated with write-induced invalidation broadcasts and collecting acknowledgments in their entirety, however it incurs area overhead by having to save a full-map vector per directory entry. At the other extreme, a hypothetical DIR_0-B would incur no area overhead, but every *write* request received at the directory would trigger an invalidation broadcast, which translates into complexity [11], [12]. Other directory techniques to optimize the area, falling in between the two extremes, result in complex directory mechanisms which significantly add to the verification cost and potentially impact scalability [9], [10].

Besides *write* requests, directory evictions might as well trigger invalidation broadcasts depending on directory policies. Today's CMPs implement directories as on-chip sparse directory-caches [8], [9], which makes them vulnerable to loss-of-information problem. Although eviction-induced invalidations can be eliminated by allocating a backing store in main memory [8], adding backing store is contrary to the goal of area-efficiency and low-complexity. As a result, inclusion is maintained between cached data and their corresponding directory entries, which consequently adds to complexity by requiring invalidations —in form of unicast, multicast, or broadcast— and acknowledgement col-

1089-795X/15 $31.00 © 2015 IEEE
DOI 10.1109/PACT.2015.23

IEEE computer society

lection [14]. Furthermore, such invalidations can potentially increase the miss rate and degrade the performance. It is still possible to eliminate the eviction-induced invalidations without requiring a backing store, however this introduces a new type of broadcast upon each directory miss in order to discover and re-build the sharing status [10].

At the other end of the spectrum, there are coherence schemes that aim to eliminate the complexities associated with the directories by removing the directories either in part or in their entirety [15], [16]. By relying on data-race-free (DRF) semantics, the need to obtain ownership upon *write* accesses is eliminated. In order to maintain data consistency under such protocols, cores perform self-invalidation (SI) and self-downgrade (SD) of their level-one (L1) cache shared data upon synchronization [17] —locks and barriers. However, such schemes are heavily software-dependent and partly delegate directory functions to other system components such as the operating system (OS), which in turn introduce new complexities and verification challenges. As an example, VIPS-M coherence protocol [16] delegates private/shared data classification at page granularity to the OS that uses the page-table in main memory as a backing store for TLB entries that hold the classification. In essence the TLBs become directory caches and the page table is the backing store. No directory information is ever lost in VIPS-M and this is one of the properties that contribute to its simplicity. In other words, it is impossible to use a (classification) directory-*cache* in VIPS without having a backing store of the whole directory or severely compromising its simplicity by introducing *broadcasts* to manage information loss from the directory [18]. DeNovo [15] is another coherence protocol which reduces the directory to track only the writers of data. However, this scheme is heavily application-dependent; furthermore, the last-level cache (LLC) is delegated to keep track of the writer for each cache line and perform request-forwarding when needed.

As the aforementioned examples show, VIPS-M and DeNovo coherence protocols partly delegate directory functionality to other system components, despite the fact that they advocate coherence simplicity by removing the directories in their entirety or in part. Both protocols require a minimum directory support to track a single owner for each piece of data. Based on this observation, we propose a new directory-based coherence scheme, called Dir_1-SISD, that bridges the gap between the conventional directory schemes [5]–[7] and the novel DRF-based coherence schemes [15], [16]. The resulting directory scheme which adopts SI and SD as directory policies, reconciles storage reduction techniques with techniques to minimize coherence complexity.

Our approach essentially performs dynamic hardware-based private/shared data classification at cache-line level: a cache line with a single sharer is considered as *private*, whereas a cache line with multiple sharers is classified as *shared*. However, unlike conventional directory schemes that enforce invalidations to maintain single-writer-multiple-readers invariant [1], our approach allows multiple-writers-multiple-readers to exist without invalidating any copies of the cache line in L1 caches. This is achieved by giving the sharers the responsibility to self-invalidate the shared data when needed. While eliminating the write-induced invalidations/broadcasts, our approach only requires tracking of a single sharer per block, which reduces the area overhead of the directory. In other words, Dir_1-SISD either tracks a single private owner or allows multiple sharers without tracking them, as long as they self-invalidate and self-downgrade. Thus, Dir_1-SISD does not require broadcasts and implements a simple coherence scheme.

Our approach also reduces the complexities associated with eviction-induced invalidations/broadcasts. Under Dir_1-SISD scheme, a directory entry may be evicted without the need to be backed-up or the need to maintain inclusion, which eliminates eviction-induced invalidations/broadcasts present in other protocols. Furthermore, as we later show in Section V, our approach enables low-complexity dual-granular directories, which further reduces area overhead by requiring a single entry per private page in the directory.

Main contributions: We propose a simple Dir_1-SISD directory organization to support self-invalidation/self-downgrade coherence that i) eliminates the reliance on the OS, page tables, and TLBs for classification, and ii) introduces no new protocol complexity such as broadcasts. This is because our proposed directory scheme has a unique characteristic not found in other directories: the on-chip directory cache is a *self-contained* directory, meaning that it neither needs to be backed-up externally nor enforces inclusion upon directory evictions. We achieve this by exploiting the self-invalidation and self-downgrade policies, as described in Section III. Further, our directory is naturally extended to multi-granular implementations as the information that is mainly tasked to track (owners of private blocks) is easily compressible to coarser granularities (e.g., regions, pages).

Why is our approach any different than what came before: While conventional Dir_1 directories either allow a single sharer or require invalidation broadcasts, our proposed scheme avoids both. We allow multiple sharers to exist without being tracked —read or write accesses— and yet we do not broadcast invalidations. Dir_1-SISD achieves this by giving the sharers the responsibility to invalidate and downgrade when the degree of sharing exceeds one. Unlike VIPS-M [16], which also uses self-invalidation and self-downgrade as its main ingredients, Dir_1-SISD does not depend on page table and TLBs to perform and store the data classification. Having a shared TLB to perform and store data classification at page granularity which is backed-up by page table upon TLB evictions, makes VIPS-M heavily OS-dependent and architecture specific. Furthermore unlike DeNovo [15], which also tracks a single private owner and employs self-invalidation, our approach does

not require indirection and request-forwarding. In other words, DeNovo only implements self-invalidation, whereas our approach implements both self-invalidation and self-downgrade. Self-downgrade obviates the need for core-to-core communication and enables implementation of simple networks that only require a two-way L1-to-LLC and LLC-to-L1 communication.

Why would our approach be of any interest to anyone: Being a SI-based protocol, Dir_1-SISD addresses architectures where DRF programming paradigm is considered as the dominant programming model. This includes all architectures programmed in modern standardized high levels languages (HLLs), such as C++11, Java, and OpenCL, where DRF *is* the prevailing model. This, however, does not mean that Dir_1-SISD cannot support data races. It certainly can do this as long as data races are intended and identified, in which case, proper fencing with self-invalidation and self-downgrade fences leads to correct code that can handle any programming construct.[1] An important aspect of Dir_1-SISD is that it does not pose any demands for OS support as previous proposals do. Dir_1-SISD is also addressed to architectures where simplicity —fast verification— and efficiency —area and energy benefits without compromising performance— might be tempting enough for designers to consider such alternatives, instead of more complex and more expensive coherence that supports sequential consistency for workloads that do not require this support.

II. Background

Censier and Feautrier *et al.* [6] were the first to propose distributed directory-based schemes in the late 70's as a solution to overcome scalability issues associated with centralized directories proposed by Tang [5]. Subsequently, Archibald *et al.* [7] and Lenoski *et al.* [19] also introduced directory techniques in which memory and directory were distributed and tiled instead of being centralized, which enabled more flexibility and scalability. Directory-based schemes have been extensively studied ever since [11], [12] and have become common in the design of CMPs. Furthermore, Gupta *et al.* [8] introduced the *coarse vector* scheme and the *sparse directory* technique to reduce directory area overhead. More recently Cuesta *et al.* [20] proposed dynamic private/shared data classification to reduce the directory area. In their approach, private data is taken out of the directory. In another recent attempt to reduce directory area overhead, Zebchuk *et al.* [21] introduced a multi-grain directory scheme which tracks sharing at *region* and cache-line granularities. All the cache lines that belong to a private region are tracked using a single *region-entry*, which results in directory area reduction, but significant complexity. In this section we briefly give a background on directory schemes that track a single sharer, which also forms the basis for our proposed Dir_1-SISD scheme. As Weber and Gupta show [22], the degree of sharing in majority of applications is very close to one and rarely goes beyond that. As a result, low complexity and overhead make Dir_1-X an appealing trade-off between resource overhead and performance.

A. Dir_1-NB

Dir_1-NB is the most basic and simplest implementation of Dir_i-X schemes [11]. It only uses one pointer per directory entry to track a single sharer at any given time. Dir_1-NB is an extreme case where a block, regardless of *read* or *write* access, is not allowed to reside in more than one core at any given time. Although restrictive, this scheme eliminates broadcasts in their entirety, which ties in closely with what directory-based coherence is trying to achieve. However, this scheme incurs higher miss rate for some workloads due to higher rate of invalidations.

B. Dir_1-B

Unlike previous scheme, Dir_1-B allows many sharers to exists simultaneously, provided that the reason for sharing is *read* access, i.e., read-only sharing [11], [12]. This results in lower miss ratio, since *readers* are not invalidated under read-only sharing paradigm. However, this capability comes at the expense of invalidation broadcasts. Since more than one sharers are allowed to exist without being tracked, a broadcast is required when a *write* request is received at the directory and more than one *reader* exist. However, as Weber and Gupta show [22], large fraction of invalidations come from *migratory* data which require a single invalidation. Although broadcasts caused by migratory sharing pattern can be eliminated by applying migratory optimization techniques, such as the one proposed by Stenström *et al.* [23], such optimizations add to the complexity and overhead of directory protocols.

C. Dir_1-SW

As a solution to complexities associated with directory-based cache coherence protocols, Hill *et al.* [24] address hardware and software issues together and propose *cooperative shared memory*. On the software side, they propose the *Check-In/Check-Out* (CICO) programming performance model. By inserting CICO annotations in the code, programmers can analyze the shared-memory communication cost. This allows the programmers to explore different design alternatives to lower the communication cost of shared-memory applications with respect to cache coherence protocols. On the hardware side, they propose a minimal directory scheme, Dir_1-SW, which delegates the complex coherence actions to software. Dir_1-SW uses a dual-purpose pointer/counter field per directory entry, which either points to a single writer, or counts the number of sharers if datum is shared by more than a single core. Upon a conflict, a trap

[1] This is no different than dealing with a relaxed memory model where fences are required for correctness, and is typically the domain of expert programmers who write library and system code.

bit is set and a software routine is invoked to resolve the coherency issue by forwarding the data to the requesting core. Programs that conform to CICO model or provide explicit CICO directives to the coherence engine run at full hardware speed, since they do not cause traps.

The essential feature of Dir_1-SW in reducing hardware-based coherence complexity is that no protocol transition implemented in hardware requires more than a single request-response message-pair and instead, software handles all the complex cases where multiple messages are required. This results in the elimination of transient states, or in other words, the elimination of hardware protocol-races, which account for the main source of complexity in hardware-based coherence protocols.

D. SCI

Scalable Coherent Interface (SCI) [25] was a comprehensive effort to address the limitations in the scalability of bus-based systems. The standard covers both physical aspects — including electrical and mechanical specifications— as well as the logical aspects —such as the coherence scheme.

To solve the coherence scalability problem, SCI proposes a directory scheme in which sharing information is distributed across the caches, implemented as a doubly-linked list. The main memory directory holds a single pointer to the *last* cache that shares a memory block. This cache becomes the *head* of the sharing list (for that block) maintained by the other caches participating in the sharing. As a Dir_1 protocol SCI deals only with the head of a sharing list, delegating the responsibility of other coherence functionality to the caches comprising the sharing list.

E. DeNovo

Similar to Dir_1-SW scheme, DeNovo architecture [15] also leverages concurrency-safety, language-level annotations. However, unlike CICO annotations which are considered as an optional performance optimization for Dir_1-SW, DeNovo architecture critically depends on a disciplined programming paradigm, such as Deterministic Parallel Java (DPJ). Choi *et al.* show that replacing *ad-hoc shared-memory* with *disciplined shared-memory* provides opportunities to design performance-, power-, and complexity-scalable hardware-based coherence schemes.

Relying on deterministic parallelism which provides data-race-freedom and explicit synchronization, an application's run-time is divided into parallel phases, with the guarantee from software that only a single thread can modify a datum in each parallel phase with no other thread accessing the datum in that phase. This obviates the need for acquiring ownership on a write, provided that each core self-invalidates, before the beginning of the next parallel phase, its L1 data that are likely to be modified by other cores.

On the other hand, *writers* should register themselves at the LLC, so that the successor *readers* know where to get the up-to-date data via request-forwarding/indirection. Therefore, each LLC entry is augmented by a pointer field (this field is smartly hidden in the space of the stale data but that is besides the point—directory functionality still exists). It is also possible that the *writer* performs a write-back before a succeeding *read* request from a new core, in which case the up-to-date data (non-stale) are present in LLC, the pointer field is not needed, and the succeeding read-requests are directly responded to by LLC.[2]

F. VIPS-M

VIPS-M [16] proposes self-invalidation and self-downgrade as the basic mechanisms for a very simple *directory-less* coherence protocol. Although advocating a directory-less coherence scheme (Dir_0-SISD), VIPS-M uses the page table as directory (and the TLBs as directory caches) to perform private-shared classification at page granularity. Cores invalidate their L1 data that belong to shared pages when the cores perform synchronization. VIPS-M eliminates request indirection *and* forwarding — present in DeNovo— by performing self-downgrade for the shared data. VIPS-M uses *write-through* as a simple self-downgrade mechanism. The majority of the write-through traffic comes from the private data, and the write-through traffic caused by shared data does not significantly add to the network traffic. As a result, a *write-back* policy is employed for private data and a *write-through* policy for shared data. VIPS-M introduces an efficient implementation of write-through by coalescing the *writes* in a write-through buffer. A core passing a synchronization point —release— forces the completion of all its pending write-throughs that may linger in the write-through buffer.

Using page table as a directory and TLBs as directory caches eliminates the loss-of-information problem when entries are evicted from the TLBs, since TLB entries are always backed-up in the page tables in memory. However, this is also one of the weak points of VIPS-M: reliance on OS support for coherence. Any attempt to break this dependency brings up problems which introduces loss-of-information problem, necessitating limitations such as enforcing inclusion via broadcasts, which may annul VIPS-M main claim of simplicity.

G. VIPS-H

VIPS-H [26] is the hierarchical implementation of VIPS-M. The similar OS-/TLB-based private/shared data classification is used to classify data across multiple clusters. VIPS-H simplifies the self-invalidation and self-downgrade in hierarchies of clustered designs by allowing a piece of data to be classified as shared among the cores within a cluster, while the same piece of data is seen as private from outside the cluster, i.e., the data is private to a cluster,

[2]DeNovo can be thought of as a Dir_1-SI protocol.

although shared within that cluster. Such classification limits the self-invalidation and self-downgrade to a single cluster and eliminates unnecessary inter-cluster communication and data movement.

While VIPS-H is optimized for and targets hierarchical clustered designs, our proposed Dir_1-SISD scheme is a base scheme which can either be used in flat organizations or can recursively be applied to hierarchical clusters. In this work we evaluate Dir_1-SISD in its base form as a flat coherence scheme, and we leave the hierarchical and clustered evaluations for future work. Another area which Dir_1-SISD differs from VIPS-H is how the data classification is handled. VIPS-H heavily relies on the OS, the page table and the TLBs for the private/shared data classification. However, Dir_1-SISD performs the classification entirely in hardware, makes it an ideal choice where interaction with OS is not feasible. Furthermore, VIPS-H performs the data classification strictly at page granularity, whereas Dir_1-SISD provides multi-granular data classification, spanning from fine cache-line granularity to coarse-grain page-based classification.

III. DIR_1-SISD

In this section we introduce our directory scheme, Dir_1-SISD, which provides a hybrid solution by coupling together the benefits of directory schemes and directory-less, DRF-based coherence protocols.

Unlike DRF-based schemes, which advocate elimination of directory in its entirety, we *do* recognize the value of having a minimal directory. We find this necessary, as this prevents the delegation of inevitable directory operations to other system components —page table, TLBs, and OS— which introduces hard-to-address complexities.

To keep both area overhead and complexity at minimum, we choose our directory to be of Dir_1 type. The key insight is that our directory essentially performs data classification to private and shared. It has no other functionality. When the directory tracks a private block we need to know the owner so we can force classification changes when needed. Otherwise, if the block becomes shared, we do not care to track the sharers. Our choice of Dir_1 is also justified by the fact that the degree of sharing is typically close to one [22]. On the other hand, in order to balance the opposing goals of area and complexity reduction, Dir_1-*SISD* takes advantage of DRF semantics to eliminate the complexities attributed to obtaining ownership or request forwarding (indirection).

A. Basic Protocol

Dir_1-SISD performs dynamic private/shared classification at cache-line granularity. By observing all the requests from all the cores in the system, our directory knows if a cache line is accessed by a single core or more cores. Each directory entry contains a single pointer which tracks the owner for private cache lines. The directory sends to the cores, along with the data responses, the classification of the data. Each core's L1 cache has a bit per cache line which stores the classification received from the directory.

Dir_1-SISD allows *multiple-readers* and *multiple-writers* to co-exist without incurring invalidations/broadcasts. Based on DRF semantics, a cache line can be shared and written by different cores simultaneously. DRF semantics guarantee that during each parallel phase of the application, *write* accesses by different cores affect different words/bytes in a cache line. Based on this paradigm, Dir_1-SISD implements *deferred invalidation* by giving the responsibility of maintaining coherence to the sharers. Instead of having the directory to invalidate the sharers, invalidation of the shared data is deferred until the end of each parallel phase marked by a synchronization, at which point each core self-invalidates its shared data [15]–[17]. Furthermore, each core self-downgrades its modified data on synchronization, making the modified data available in a shared LLC to other cores to access during the next parallel phase. Self-downgrade is performed via an efficient implementation of write-through policy for shared data proposed by our previous work [16]. In that implementation, *writes* are delayed in a write-through buffer to allow coalescing. Because there may exist multiple simultaneous writers for a cache line (as allowed by DRF semantics), each modifying a different part of the cache line, a write-through must communicate only the modified data to the shared cache, i.e., send a *diff* of the cache line since its last write-through. For this reason per-byte dirty-bits are used but these bits are only needed while a cache line is in the write-through buffer and is actively modified. Each cache line also carries a single line-dirty bit that is used to perform write-back for private data.

B. Private-to-Shared Transition

Dir_1-SISD is tasked to track the owners of private blocks, and in the case of shared blocks to distribute the responsibility for self-invalidation and self-downgrade to the cores that share the block. Thus, measures need to be taken when Dir_1-SISD receives a request for a private cache line, with the requestor being different from the registered owner. At this point, directory notifies the private owner to change its local classification for the cache line from private to shared. This notification, which we refer to as *recovery* [20], is in the form of a unicast, which simplifies the network design. In addition, the Dir_1-SISD entry for the cache line transitions *shared* and further accesses by other cores are responded as such.

Upon receiving a *recovery* notification, a core may respond in one of the following ways:

1) The core responds with a NACK message if it has already evicted the cache line. This enables the directory to maintain the private classification for the block, however with a new owner.

2) The core changes its classification for an unmodified cache line from private to shared. This causes the cache line to be self-invalidated at the next synchronization performed by the core. The core then responds with a *clean* ACK message, resulting in the cache line being globally classified as shared by the directory. The directory then responds to the new requestor with shared data.
3) For a modified private cache line, the core goes through the same steps as for a clean cache-line, however the core responds to the directory with a *dirty* ACK which includes the modified (whole) cache line. This is necessary so that the new requestor receives the up-to-date data.

We address the cost of recovery in section VII-B2.

IV. Directory Eviction, Self-Correction, and Adaptation

Besides write-induced coherence actions, actions triggered upon directory evictions also impact the complexities associated with directories. Today's large memory sizes make it impractical to provide full directory coverage and implement in-memory backing stores. As a result, measures need to be taken to address the loss-of-information problem when directory entries are evicted. Maintaining inclusion between directory and L1 caches is a common practice in the design of Dir_1 protocols. In fact, L1–Directory inclusion has also been used in directory caches for *full-map* directories, as in the case of the SUN Microsystems *Sunfire* system (Enterprise 6000). This certainly obviates the need for a backing store, however invalidations for inclusion potentially degrade the performance and require broadcast support from the network. While silent eviction of directory entries without a backing store is possible, this policy introduces a new type of broadcast upon each directory miss, needed to discover and re-build the sharing status [10], [18].

Reliance on DRF semantics provides a feature unique to Dir_1-SISD which, without a need for backing store, allows the directory entries to be evicted without maintaining inclusion. This is achieved by having the cores to maintain the coherence, allowing the directory information to be discarded without invalidating the corresponding cache lines.

As discussed in section III-A, no directory-side coherence actions are invoked after data is classified as shared. Cores perform self-invalidation and self-downgrade based on DRF semantics, which guarantees data consistency for the shared data. In other words, the role of Dir_1-SISD directory in maintaining coherence is only limited to private data and handling the private-to-shared transition via the *recovery* action. Coherence for the shared data is maintained via core-side coherence actions, which include self-invalidation and self-downgrade. Based on these properties, shared directory entries can be evicted silently. For the private directory entries, the responsibility of maintaining coherence is transferred to the cores before the directory entry can be evicted. The process of forcing the classification of a cache line from private to shared upon directory eviction does not add to the complexity of the directory by introducing new coherence mechanisms and actions. The *force-sharing* process is the same as the *recovery* process described in section III-B, which is already implemented. We discuss the cost of force-sharing in section VII-B2.

To reiterate: shared directory entries are silently evicted, private directory entries are converted to shared and then silently evicted. In other words, we disguise private cache lines as shared to avoid their immediate eviction. Of course, private cache lines turned-to-shared are destined to be self-invalidated at the next synchronization point of their corresponding core, however the lease-of-life that this mechanism allows is significant and makes a difference for Dir_1-SISD as we will show in section VII.

A. Self-Correcting Classification

We are now faced with the situation where shared cache lines *and* private cache lines (*intentionally misclassified as shared*) can exist without a corresponding directory entry. What happens if such a directory entry is created again as a result of an L1 miss in some other core? A new directory entry is always created as *Private*, since we have no other knowledge. The *current* private owner, assumes that it is alone and refrains from self-invalidating and self-downgrading, even in the presence of other shared copies — genuinely shared or private-turned-shared— in the system. However, this is *allowed* by DRF semantics and Dir_1-SISD *self-corrects* the classification as follows.

- Based on DRF semantics, a cache line can simultaneously be shared and written by multiple sharers, since software guarantees that cores do not access same bytes in a cache line during each parallel phase. Thus, even when the current private owner modifies the data but fails to self-downgrade them on synchronization, no harm is done since the modifications could remain invisible to any other core until that core passes a synchronization point and self-invalidates. After the synchronization, the modified data can be accessed via the *recovery* process explained in section III-B, as the directory holds the pointer to the private block. The private owner is then corrected to shared.
- On the other hand, it is possible that a shared copy self-downgrades while another core has received the cache line as private. Again, based on DRF semantics, data consistency is not violated. The private owner is notified, via a *recovery* notification (see Sec.III-B) to change the classification to shared, which means that it will self-invalidate the line on its next synchronization point after which it will be able to access the updated data. The critical observation here is that the transition

from private to shared *happens before* the private owner passes the synchronization point that makes the updated data visible. This is enforced by DRF semantics: the writer (shared copy) and the reader (private copy) must be in a happens-before release-acquire order via their synchronization. Thus, the private-to-shared transition must happen before the release, which guarantees that the private copy now corrected to shared *cannot pass the acquire point without self-invalidating*. Again, the correction of classification does not impose complexities on the directory by requiring new actions, since the already existing *recovery* mechanism is reused.
- If none of the above two cases occur, then the co-existence of a private and one or more shared copies at the same time is harmless: either there is no communication among them, or the shared copies eventually self-invalidate on synchronization leaving the private copy truly alone.

This self-correcting behavior of the Dir_1-SISD, is enabled by a simple classification invariant: *it is impossible for the same block to be private at different caches, although a single private copy and several shared (SISD) copies is allowed.* We discuss the cost of self-correction in section VII-B2 and show that self-correction is a rare event whose cost is negligible.

B. Adaptive Classification through Directory Replacement

Having fewer data classified as shared mitigates the penalty incurred by self-invalidation of shared data. Adaptive data classification, in which a shared piece of data can be reclassified as private, becomes critical in the performance of systems based on self-invalidation. Alisafaee [27] proposes *spatiotemporal coherence tracking,* in which a shared piece of data can temporarily be considered as private. However, such proposals require complex mechanisms that nullify the benefits.

Directory eviction in Dir_1-SISD provides a natural means for adaptive classification with zero overhead, allowing the next requestor to classify the data as private, since the address is not found in the directory. As discussed in the previous section, if the requestor is misclassified as private the Dir_1-SISD invariant and DRF semantics will soon correct the classification. If however, a shared block transitions into a period where it becomes private, then the shared copies of the block will disappear by self-invalidation, leaving the requestor as private. We emphasize again that this is *without cost.*

One can go a step further and use this Dir_1-SISD capability to manage the adaptivity rate by manipulating the eviction rate of shared directory entries, for example by giving preference to evicting shared entries, or by decaying them after a period of inactivity, or even by appropriately sizing the directory itself at run-time. While we did not explore active manipulation of the eviction rate, our evaluation incorporates the natural adaptivity that results from a limited directory size. We leave the more advanced techniques for future work.

C. Thread Migration

A persistent problem in private/shared classification is thread migration. Upon a thread migration, private data become shared as they are now accessed from a different cache. Thus, we end up self-invalidating and self-downgrading thread-private data at great cost. Solutions have been proposed for page-level classification [28] (essentially flushing the private data and restarting the classification after the migration), however these solutions add complexity and become expensive in the case of frequent and intensive migration. Adaptive classification on the other hand manages thread migration by default as it turns shared data into private. In the case of Dir_1-SISD, this adaptation is costless and comes from the eviction of the shared entries that correspond to the thread private data. Distinguishing between CPU and GPU domains helps to better address the thread migration issue:

- *CPU domain.* Thread migration is not a common case in CPU domain. Therefore, the shared data misclassification is not harmful even if left untreated. However, there are a variety of techniques that can be used to predict the dead cache-lines, and therefore maintain the private data classification. As an example, cache decay techniques [29], [30] can be used to detect the dead blocks, resulting in a NACK to be sent in response to a recovery (see III-B), enabling re-classification of data as private after thread migration. Furthermore, we have shown and evaluated how cache decay can be used as a simple dead-block predictor in order to maintain private classification upon recoveries [31]. Cache decay has also been employed in TLBs seeking a more accurate classification at page granularity [32], [33].
- *GPU domain.* Thread migration is more common in GPUs. However, experiments reveal that replacement rate of blocks in GPU caches is so high, which guarantees that recoveries return NACKs with a high probability and therefore private classification is maintained even without the need to employ dead-block predictors.

V. Directory Compression

In contrast to other directories, the primary functionality of Dir_1-SISD is to track private data and their owners, rather than shared data. This is evident by the ease of evicting shared directory entries that carry very little information (simply that the line is shared) and their subsequent replacement by private entries. However, private data are far more common than shared data which means that Dir_1-SISD may face increased pressure compared to directories that aim to track only shared data [20]. The realization that addresses this issue is that, although private data are far more numerous

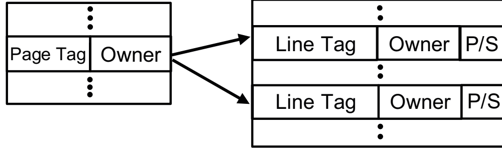


Figure 1. Logical organization of a dual-grain directory implementation. A page directory —on the left— keeps track of a page owner. Shared cache lines within a private page, or private cache lines with an owner different than the page owner are tracked by a line directory —on the right. Page directory is chosen to be smaller than line directory, as it compresses all the page's tags into a single entry.

than shared, their directory information can be "*compressed*" far more easily [21], [27] than the corresponding directory information of shared data [34]. This leads to region directories or multi-granular directories where private data are tracked at coarser granularities (typically the page size works best).

A. Case-study: dual-grain directory

Although dual-grain directories have been proposed in the past, such organizations require complex directory mechanisms. As an example, Zebchuk *et al.* introduce the dual-grain *DGD* directory [21] which allows both region and block entries, each with a different format, to reside in the same directory cache. However, in order to minimize directory lookup latency, DGD deserializes region and block entry lookups by imposing a restriction on how the entries are mapped in the directory cache: the region entries are required to be mapped to half of the ways, while the block entries are mapped to the other half. This allows region and the corresponding block entries to be found in a single lookup, however this reduces lookup associativity. They further try to mitigate this problem by employing replacement policies aimed to minimize directory evictions, however such replacement policies are based on more complex hashing techniques [35], [36].

Dir_1-SISD, however, lends itself very well to low-complexity dual-grain directory organizations. In this organization, as shown in Fig.1, the directory is composed of a *line-directory* and a *page-directory* (both operating as Dir_1-SISD with the only difference being the granularity). All the cache lines belonging to a private page are represented by a single entry in the page-directory. In a system with 64-line pages, this translates into compression ratio of $1 \div 64$, which significantly reduces the directory area. Furthermore, page-directory and line-directory are looked up simultaneously, without incurring complexities associated with DGD.

What is important in this case is the interaction between the two directories. First access to a page causes an entry in the page-directory to be allocated. Therefore, the first core accessing a page becomes the private owner of that page. Further accesses by the page owner do not change the directory state. Upon receiving an access from a core other than the page owner, an entry in line-directory is allocated to resolve the classification for the conflicting cache line. While the entries in the page-directory always point to private owners —the first core accessing a page,— entries in line-directory might be classified as either private or shared. After a conflict, a *recovery* (Sec.III-B) is performed to decide if the cache line should be classified as private or shared in the line-directory. If the recovery is successful, i.e., the specific block being accessed by a new core also exists in the private owner's cache then both become shared and the entry in the line-directory starts as shared. If, however, the line is *not found* in the cache of the page-private owner, then it starts as private (with the core that accessed the line as the new owner) in the line-directory.

This decision could be taken without performing a recovery, if we know which lines belong to the private owner. In other words, if we enhance the page-directory entries with a bit-map for the lines in the page, we can discern which lines belong to the private page and which do not, simply by accessing the page-directory entry. However, this adds cost and slightly increases the complexity of handling page-directory entries and we do not use it.

Line-directory entries that revert to the same owner as the their page, are folded back to the corresponding page only via eviction, similarly in philosophy to the adaptation discussed above. The benefit is that folding back to the page-directory entry comes for free, albeit after the end of the lifetime of a line-entry. There is no other mechanism to support this functionality.

Evictions (due to a replacement) from the page-directory are correspondingly more expensive than line-directory evictions. A *page* recovery concerns all the lines belonging to a page that are resident in the cache of the private owner. All such lines must be changed from private to shared. Note that we do not need to install the corresponding entries in the line-directory, as the Dir_1-SISD concept allows shared lines without a corresponding directory entry. In this case also, a line bit-map in the page-directory entries can be used to avoid evicting pages with many private cache lines and lessen the overhead. Such a bit-map would allow us to set a threshold of private lines in a page, *under* which the entry can be selected for replacement. While this optimization in the replacement policy can reduce the cost of evicting page-directory entries (at the cost of increased storage requirements and increased complexity), in practice it may not be needed if the page-directory has a wide-enough coverage. We leave this study for future work.

Fig.2 shows the steps involved in a directory look-up for our proposed dual-grain directory implementation. In section VII-B5 we evaluate the opportunities to compress the directory using a dual-grain organization.

VI. Directory Organization

In this section we describe the directory organizations that we use in our evaluation.

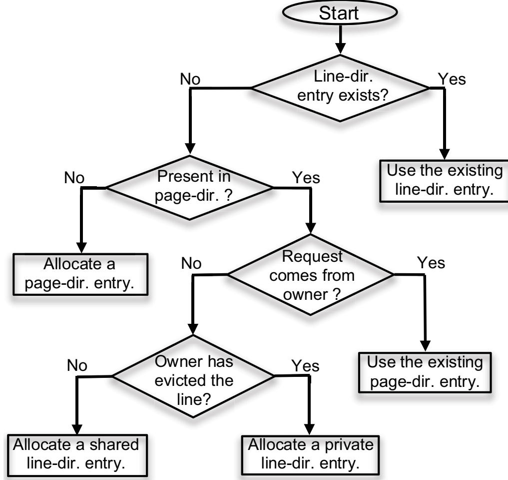
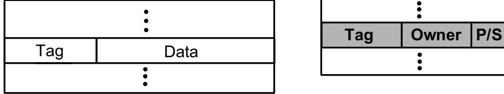


Figure 2. Dual-grain directory look-up routine.

(a) Coupled LLC-directory implementation: LLC tags are extended to hold private/shared classification. The *owner* field is valid when a cache line is classified as private.



(b) Decoupled LLC-directory implementation: directory area is reduced —on the right— by having a directory cache independent of LLC size.

Figure 3. Logical organization of Dir1_1-SISD directory.

A. In-Cache Directory.

As a simple organization, directory and LLC can be coupled together [37], [38], referred to as *static-bank-directory*. As shown in Fig.3a, each LLC tag is augmented with a directory field which holds the classification for that cache line. In this organization, an LLC entry replacement forces the replacement of the corresponding directory entry and vice versa. As a result, LLC and directory misses are equivalent, which requires a simpler controller design.

B. Stand-Alone Directory Cache.

Despite its simplicity, the in-cache directory organization incurs area overhead: each LLC tag should be augmented with a directory field, whereas only a subset of all those entries are required at any moment in time to perform directory actions for the data cached by the cores in the system. The area overhead becomes pronounced with the increasing size of on-chip LLC. To address this problem, a directory cache —independent of LLC size— can be used [8]. In this organization, LLC replacements are decoupled from the directory replacements. Although the resulting organization requires a more complex controller design, the area is significantly reduced (Fig.3b). As an example, for a system configuration with 32 KB L1 cache and 512 KB LLC cache per tile, a stand-alone organization only requires 64K directory entries with overprovisioning factor of two, whereas an in-cache organization requires 512K directory entries, eight times more than the stand-alone organization.

Table I
BASE SYSTEM PARAMETERS

Memory Parameters	
Processor frequency	3.0GHz
Block size	64 bytes
MSHR size	16 entries
Split L1 I & D caches	32KB, 4-way
L1 cache hit time	1 (tag) and 2 (tag+data) cycles
Shared unified LLC cache	8MB, 512KB/tile, 16-way
LLC bank cache hit time	6 (tag) and 12 (tag+data) cycles
L1-LLC inclusion policy	Inclusive
MESI Directory	Full-map in LLC tags
Memory access time	160 cycles
Page size	4KB (64 blocks)
Network Parameters	
Topology	2-dimensional mesh (4x4)
Routing technique	Deterministic X-Y
Flit size	16 bytes
Data and control message size	72 bytes (5 flits) data, 8 bytes (1 flit) control
Routing, switch, and link time	2, 2, and 2 cycles

VII. Evaluation

A. Setup

We evaluate Dir_1-SISD (DRF semantics) against a Dir_n-NB protocol (MESI states) and also the VIPS-M (DRF semantics) protocol which operates at page granularity with the involvement of the OS, page table, and TLBs. We also compare Dir_1-SISD against an inclusive, adaptive, DRF-based protocol which operates at cache-line granularity — referred to as *Adaptive*, explained in section VII-B3— in order to study the classification adaptation property of Dir_1-SISD. Our target system is a 16-tile chip multiprocessor. We use the Simics full-system simulator [39], and model VIPS-M, Adaptive, and Dir_1-SISD protocols using the cycle-accurate GEMS simulator [40]. We also employ the GARNET network simulator [41] to model the interconnection network. Furthermore, we model the target system using Pin [42] in order to study the opportunities to compress the directory using a dual-grain directory implementation (Sec. VII-B5). Table I gives the main parameters of our base system.

We employ a wide variety of parallel applications. *Barnes* (16K particles), *FFT* (64K complex doubles), *FMM* (16K particles), *LU-CB* (512×512 matrix), *Ocean* (514×514 ocean, contiguous partitions), *Radiosity* (room, -ae 5000.0 -en 0.050 -bf 0.10), *Raytrace* (teapot, optimized version that removes unnecessary locks), *Volrend* (head), *Water-Nsq* (512 molecules) and *Water-Sp* (512 molecules) belong to the *SPLASH-2* benchmark suite [43]. Tomcatv (256points, 5 time steps) is a shared-memory implementation of the SPEC benchmark [44]. *Blackscholes* (simsmall), *Canneal* (simsmall), *Swaptions* (simsmall), and *x264* (simsmall) are from the PARSEC benchmark suite [45]. We simulate the entire applications, but collect statistics only from start to completion of their parallel part.

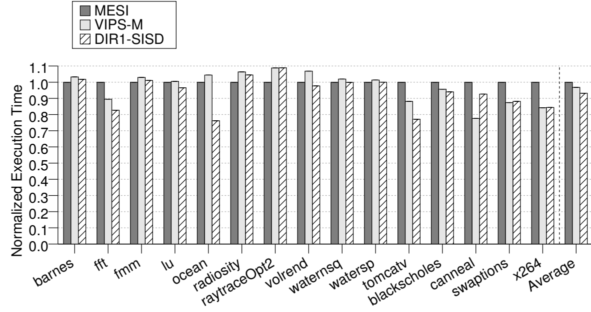


Figure 4. Dir_1-SISD performance comparison. Results are normalized to MESI protocol. Dir_1-SISD is implemented as an in-cache directory.

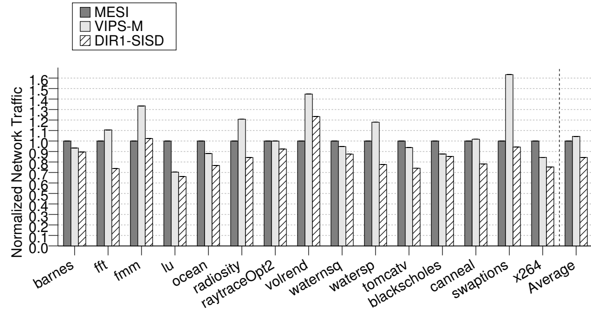


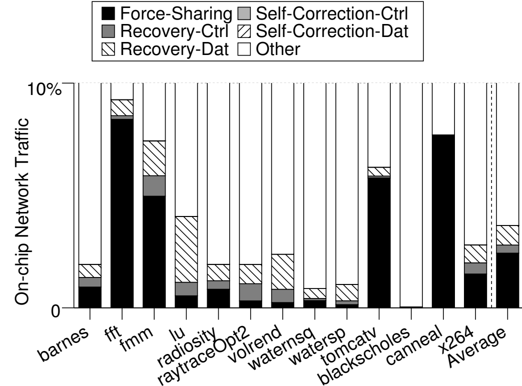
Figure 5. Dir_1-SISD network traffic comparison. Results are normalized to MESI protocol. Dir_1-SISD is implemented as an in-cache directory.

B. Results

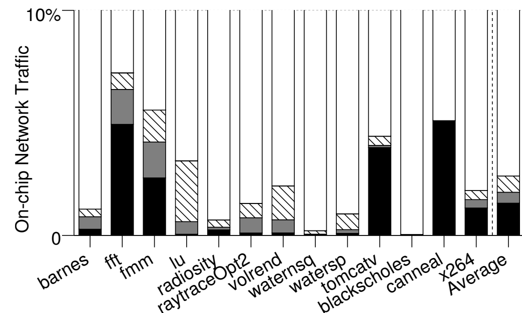
1) DIR_1-SISD vs. MESI vs. VIPS-M: Fig.4 and Fig.5 show the comparison of MESI (Dir_n-NB), VIPS-M, and Dir_1-SISD protocols with respect to performance and network traffic. Dir_1-SISD and MESI are implemented as an in-cache directories, shown in Fig.3a.

As Fig.4 shows, Dir_1-SISD achieves a better performance on average. *Canneal* is the only benchmark which Dir_1-SISD performs significantly worse than VIPS-M. The performance loss is due to the fact that *canneal* is mostly comprised of private data, with a very low amount of sharing. As a result, the majority of directory evictions invoke the recovery mechanism discussed in section IV. However, VIPS-M is not vulnerable to loss-of-information problem. Since classification information is always backed up by the page table in main memory, VIPS-M silently replaces unmodified TLB entries —Private TLB entries.

Dir_1-SISD reduces the network traffic by about 12% on average compared to MESI, as depicted by Fig.5. The MESI protocol provided by GEMS is implemented using a full-map directory, which eliminates the need for broadcasts upon directory evictions and write misses at the expense of directory area overhead. Even in the absence of broadcasts, Fig.5 shows that Dir_1-SISD reduces the traffic compared to MESI. This is mainly due to the elimination of write-induced invalidations. We expect to see more reduction in the network traffic when Dir_1-SISD is compared to a non-full-map implementation of MESI protocol. On the other hand, both VIPS-M and Dir_1-SISD incur more network traffic in *raytrace* and *volrend* benchmarks. This is explained by referring to the larger amount of shared data and frequent synchronizations in those benchmarks, which result in invalidation of the shared data. The self-invalidated data is accessed frequently in those benchmarks, which incurs data movement due to frequent cache misses and also self-downgrade for the shared data in the form of write-through traffic.



(a) Number of directory entries equals number of L1-cache entries.



(b) Number of directory entries is twice the number of L1-cache entries.

Figure 6. Network traffic breakdown. The graph shows the traffic pertaining to recovery (Sec. III-B), force-sharing (Sec. IV) and self-correction (Sec. IV-A). The Y-axis shows up to 10% of the total network traffic, since the rest of the network traffic is only composed of the *other* component shown in white color.

As depicted in Fig.5, Dir_1-SISD is consistent in reducing the network traffic compared to VIPS-M. This is explained by referring to the granularity at which the two protocols operate. VIPS-M inclines to misclassify cache lines as shared. This happens when a page is shared among threads for only a few cache lines within the page. This results in the whole page to be classified as shared, since VIPS-M performs data classification at page level. On the other hand, more shared data translates into more self-invalidation, which incurs extra traffic if the invalidated data is reaccessed frequently.

2) Recovery, directory replacement, and self-correction cost: Fig.6a shows the network traffic associated with

recovery (Sec. III-B), force-sharing (Sec. IV) and self-correction (Sec. IV-A) discussed earlier. As the figure shows, the majority of the on-chip network traffic—all the traffic from 10% of the total traffic up to the 100%, which is not shown in the graph—pertains to regular GET and PUT requests and the data transfers associated with them (referred to in the figure as *other*). Traffic due to self-correction (Sec. IV-A) is negligible. This traffic is so low that it cannot be distinguished in the figure. This confirms that self-correction is a rare event. Figure also shows that recovery (Sec. III-B) corresponds to 1.4% of the total network traffic on average. Recovery traffic is composed of *control* and *data* components. The *data* component is associated with the modified private blocks in L1 caches that are downgraded in response to the recovery requests from the LLC. Recovery requests and non-data responses (clean ACK or NACK) form the *control* component of recovery. According to Fig.6a directory replacement, which we refer to as force-sharing (Sec. IV), is more costly than the other two types of recoveries, however it still has a low traffic contribution, equal to 2.3% of the whole on-chip traffic. Force-sharing is only composed of a *control* component, i.e., L1 caches never return data to a directory replacement request, but only inform the directory via a control message that the classification of the block is internally changed to shared. It is still possible to alleviate the cost of directory replacement by increasing the directory coverage. As an example, Fig.6b shows that the network traffic caused by directory replacements is reduced to 1.4% by increasing the directory coverage from 1x to 2x.

3) Data classification adaptation: As discussed in section IV-B, Dir_1-SISD provides adaptive shared-to-private classification as an intrinsic feature at no extra cost. This appealing side effect is the result of directory evictions, which allows the subsequent request to classify the data as private.

Coherence schemes have been proposed which provide shared-to-private classification adaptation by allowing the shared data to be temporarily classified as private [21], [27]. Such schemes require mechanisms to explicitly perform adaptation. To evaluate the efficiency of Dir_1-SISD with respect to data classification adaptation, we implement an adaptive DRF-based coherence protocol at cache-line granularity. The *Adaptive* coherence protocol performs private/shared classification in the LLC based on the observed requests, similar to Dir_1-SISD. However, *Adaptive* maintains inclusion between LLC and the L1s, which requires invalidations —unicast or broadcast, depending on the private/shared classification of the evicted entry— to be sent to the L1s. On the other hand, cores explicitly notify the LLC when they replace cache lines. Unlike other adaptive proposals which require explicit replacement notifications for all the cache lines [1], [21], [27], we optimize *Adaptive* so that replacement notifications are only required for the shared data. This optimization significantly reduces the network traffic overhead associated with replacement notifications.

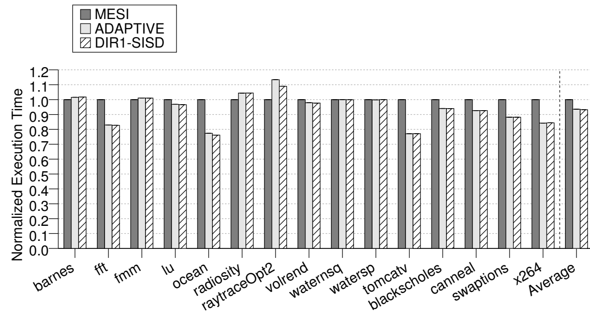


Figure 7. Dir_1-SISD performance compared to an adaptive DRF-based protocol. Normalized to MESI protocol.

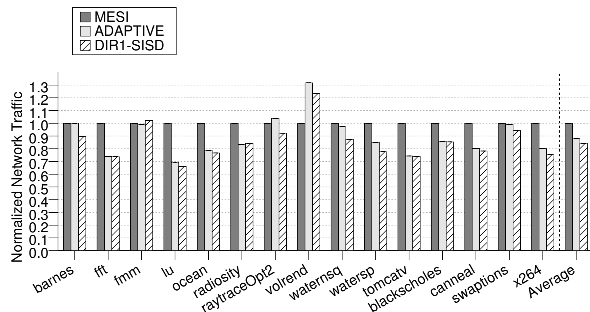


Figure 8. Dir_1-SISD network traffic compared to an adaptive DRF-based protocol. Normalized to MESI protocol.

By tracking the number of requests for each cache line and the number of received eviction notifications, LLC can detect when a cache line transitions from shared to private.

As depicted in Fig.7 and Fig.8, Dir_1-SISD performs slightly better than *Adaptive* protocol and also reduces the network traffic. The reduction in the network traffic is explained by addressing the notification messages sent by L1s to LLC upon each L1 eviction. Although the *Adaptive* protocol results in more data classified as private, the impact on performance is negligible due to the fact that the private data is not reused. Dir_1-SISD, however, results in more private data that are in active use, without incurring eviction notification from the L1s. In other words, the *Adaptive* protocol and the similar proposals perform shared-to-private adaptation based on the information received when the life-time of data is finished. This requires that cores notify the classification mechanism of the end of data life-time, which in some cases results in private data that is not reused. Dir_1-SISD, on the other hand, begins new private classification at the beginning of the life-time of data, which is already signaled by data requests received from the cores, and when the corresponding directory entry is already evicted from the directory due to its inactivity without requiring any extra information to signal the eviction. The latter guarantees that the private data in the system is in active use, therefore skips the overhead of private reclassification when the lifetime of data has ended.

Fig.8 also shows that Dir_1-SISD results in slightly more network traffic than *Adaptive* protocol in *FMM* benchmark.

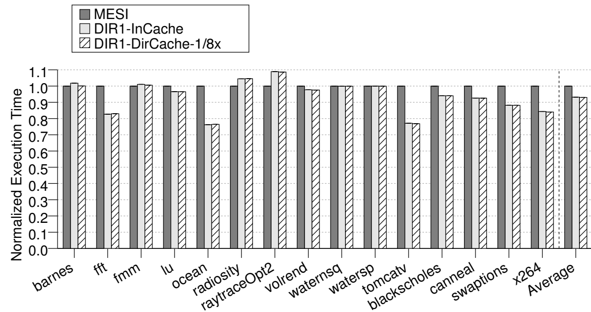


Figure 9. In-cache vs. directory-cache performance comparison. Normalized to MESI protocol.

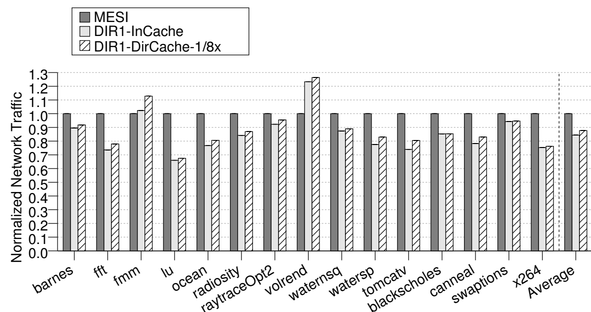


Figure 10. In-cache vs. directory-cache network traffic comparison. Normalized to MESI protocol.

This is explained by referring to the replacement policy employed by Dir_1-SISD. Based on *Least Recently Used* (LRU) replacement policy, the private entries in the directory are prioritized for replacement. This is due to the fact that private data follow write-back policy —as opposed to write-through policy for shared data— which results in private entries of directory receiving less activity. Dir_1-SISD forces the private data in L1s to shared upon directory evictions, which increases the amount of shared data in the system, resulting in increased network traffic in the form of write-through traffic. To remedy this counter-adaptation effect, the replacement policy can be modified to prioritize the shared entries over the private ones for eviction from directory. Such a replacement policy, besides eliminating the undesired private-to-shared adaptation, also allows silent directory replacements by eliminating the recovery operation associated with private entry evictions.

4) In-cache vs. stand-alone directory-cache: We also compare the two directory organizations discussed in section VI with respect to performance and network traffic, as shown in Fig.9 and Fig.10, respectively.

We configure the directory cache to have eight times fewer entries than the in-cache directory implementation. Each LLC tile consists of 512 sets, each set being 16-way set-associative. Therefore, our directory cache has 128 sets, each set being 8-way set-associative. Since L1 instruction and data caches are 128-sets 4-way each, our directory cache has entries equal to the aggregate number of entries in instruction and data caches.

As Fig.9 depicts, it is possible to maintain the same performance by reducing the number of directory entries eight times. Decoupling the directory from LLC allows large LLCs to exist without incurring directory overhead. Furthermore, Fig.10 shows that, on average, the network traffic is only slightly increased outside the critical path. The static-power savings due to directory area reduction by the factor of eight makes the slightly increased network traffic an acceptable trade-off. Moreover, the slight increase in the network traffic can be further mitigated by choosing directory area reduction factor of 4 or 2, instead of 8.

5) Area and Directory-Compression: It is self-evident that Dir_1-SISD has an area advantage over a Dir_n-NB for the same number of directory entries. Specifically, the pointer of a Dir_1 scales with system size N as $\log_2$(N), whereas the bit-map of a Dir_n only as N, offering the corresponding area savings. More interesting however is the case of a stand-alone directory cache implementation: whereas *the number of entries* of the Dir_n-NB should be over provisioned by 2x with respect to the cache entries because of the inclusion property [8], we do not have such a need because we do not enforce inclusion. (The alternative for Dir_n-NB is to resort to highly-complex directory cache implementations [9], [21].) This unique ability allows us to reduce the size of our directory cache by 8x without perceptible impact on performance. In terms of total directory storage requirements, a page-based directory reserves a pointer for all allocated pages, and caches this information in the TLBs, so its hardware storage requirements are a function of the total number of TLB entries. In terms of hardware area overhead this is comparable to our stand-alone Dir_1-SISD implementation *and* we do not require any backing storage. In addition to the inherent area advantages of the Dir_1-SISD, a multi-granular approach can be used to further reduce its area requirements for a given coverage or expand the coverage for a given number of directory entries.

To investigate the opportunities to compress the directory, we model a dual-grain directory discussed in Sec. V using Pin tools [42]. We model a 16-core system with 128-sets 4-ways L1 caches. We model 128-sets 16-ways line-directory, which is the aggregate capacity of the L1 caches. We model unlimited-size directories, which allows us to get insight into true compression opportunities inherent in the applications without the obfuscating effects of directory replacement. Fig.11 shows the percentage of the line-directory allocations that can be eliminated in presence of a page-directory. Overall, on average for this set of benchmarks, total line-directory allocations can be reduced by a significant factor of 71.38%.

VIII. Conclusion

In this work we introduce a new directory, Dir_1-SISD, that uses self-invalidation and self-downgrade as its directory policies. It tracks the private owner of a line or distributes

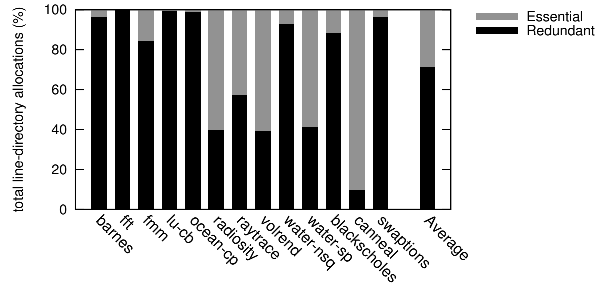


Figure 11. Opportunities to compress the directory. Line-directory area can be compressed by eliminating the redundant entries. Redundant entries are compressed in a much smaller page-directory.

the responsibility of coherence to the cores if the line becomes shared (accessed by more than one). This functionality allows for some remarkable properties. It allows us to build *self-contained directory caches* with the unique ability of neither requiring a backing store nor enforcing inclusion with cached lines. With correctness guaranteed by DRF semantics, Dir_1-SISD allows the temporary coexistence of a *single* “private” line and multiple shared lines and subsequently *self-corrects* the classification. It achieves this without burdening the underline SISD protocol with any additional complexity, while at the same time breaks the reliance to the OS, page tables, and TLBs that previous proposals have. We show that it performs better classification than the OS at page granularity and than hardware at line granularity with inclusion. Dir_1-SISD naturally adapts from shared to private via its directory evictions and does this better than an adaptive classification that tracks cache line evictions. Finally, Dir_1-SISD can be straightforward extended to multi-granular approaches without incurring any additional protocol complexity.

ACKNOWLEDGMENT

This work was supported in part by the Swedish VR (grant no. 621-2012-5332), Vinnova Vinn-Verifiering (award: VIPS 2013-01113), ”Fundación Seneca-Agencia de Ciencia y Tecnología de la Región de Murcia” under grant ”Jóvenes Líderes en Investigación” 18956/JLI/13, and by the Spanish MINECO, as well as European Commission FEDER funds, under grant TIN2012-38341-C04-03.

REFERENCES

[1] M. M. K. Martin, M. D. Hill, and D. J. Sorin, “Why on-chip cache coherence is here to stay,” *Communications of the ACM*, vol. 55, pp. 78–89, Jul. 2012.

[2] J. R. Goodman, “Using cache memory to reduce processor-memory traffic,” in *10th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 1983, pp. 124–131.

[3] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, “Implementing a cache consistency protocol,” in *12th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 1985, pp. 276–283.

[4] M. S. Papamarcos and J. H. Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” in *11th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 1984, pp. 348–354.

[5] C. K. Tang, “Cache system design in the tightly coupled multiprocessor syste,” in *AFIPS 76, 10th national computer conference and exposition, NY*, Jun. 1976, pp. 749–753.

[6] L. M. Censier and P. Feautrier, “A new solution to coherence problems in multicache systems,” *IEEE Transactions on Computers (TC)*, vol. 27, no. 12, pp. 1112–1118, Dec. 1978. [Online]. Available: citeseer.ist.psu.edu/context/1651/0

[7] J. Archibald and J. L. Baer, “An economical solution to the cache coherence problem,” in *12th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 1985, pp. 355–362.

[8] A. Gupta, W.-D. Weber, and T. C. Mowry, “Reducing memory and traffic requirements for scalable directory-based cache coherence schemes,” in *Int’l Conf. on Parallel Processing (ICPP)*, Aug. 1990, pp. 312–321.

[9] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, “Cuckoo directory: A scalable directory for many-core systems,” in *17th Int’l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2011, pp. 169–180.

[10] S. Demetriades and S. Cho, “Stash directory: A scalable directory for many-core coherence,” in *20th Int’l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2014, pp. 177–188.

[11] A. Agarwal, R. Simoni, J. L. Hennessy, and M. A. Horowitz, “An evaluation of directory schemes for cache coherence,” in *15th Int’l Symp. on Computer Architecture (ISCA)*, May 1988, pp. 280–289.

[12] D. A. Wood, S. Chandra, B. Falsafi, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, S. S. Mukherjee, S. Palacharla, and S. K. Reinhardt, “Mechanisms for cooperative shared memory,” in *20st Int’l Symp. on Computer Architecture (ISCA)*, May 1993, pp. 156–167.

[13] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture, M. D. Hill, Ed. Morgan & Claypool Publishers, 2011.

[14] E. Hagersten and M. Koster, “Wildfire: A scalable path for SMPs,” in *5th Int’l Symp. on High-Performance Computer Architecture (HPCA)*, Jan. 1999, pp. 172–181.

[15] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, “Denovo: Rethinking the memory hierarchy for disciplined parallelism,” in *20th Int’l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011, pp. 155–166.

[16] A. Ros and S. Kaxiras, “Complexity-effective multicore coherence,” in *21st Int’l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 241–252.

[17] A. R. Lebeck and D. A. Wood, “Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors,” in *22nd Int’l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 48–59.

[18] S. Kaxiras and A. Ros, "Efficient, snoopless, soc coherence," in *25th IEEE International System-on-Chip Conference (IEEE SOCC)*, Sep. 2012, pp. 230–235.

[19] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. A. Horowitz, and M. S. Lam, "The stanford DASH multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, Mar. 1992.

[20] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *38th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2011, pp. 93–103.

[21] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-grain coherence directories," in *46th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2013, pp. 359–370.

[22] W.-D. Weber and A. Gupta, "Analysis of cache invalidation patterns in multiprocessors," in *3th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Apr. 1989, pp. 243–256.

[23] P. Stenström, M. Brorsson, and L. Sandberg, "An adaptive cache coherence protocol optimized for migratory sharing," in *20st Int'l Symp. on Computer Architecture (ISCA)*, May 1993, pp. 109–118.

[24] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, "Cooperative shared memory: Software and hardware for scalable multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 11, no. 4, pp. 300–318, Nov. 1993.

[25] D. B. Gustavson, "The scalable coherent interface and related standards proyects," *IEEE Micro*, vol. 12, no. 1, pp. 10–22, Jan. 1992.

[26] A. Ros, M. Davari, and S. Kaxiras, "Hierarchical private/shared classification: The key to simple and efficient coherence for clustered cache hierarchies," in *21st Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 186–197.

[27] M. Alisafaee, "Spatiotemporal coherence tracking," in *45th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2012, pp. 341–350.

[28] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: Near-optimal block placement and replication in distributed caches," in *36th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2009, pp. 184–195.

[29] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *28th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2001, pp. 240–251.

[30] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *28th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2001, pp. 144–154.

[31] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras, "The effects of granularity and adaptivity on private/shared classification for coherence," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 3, Oct. 2015. [Online]. Available: http://dx.doi.org/10.1145/2790301

[32] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato, "Temporal-aware mechanism to detect private data in chip multiprocessors," in *42nd Int'l Conf. on Parallel Processing (ICPP)*, Oct. 2013, pp. 562–571.

[33] A. Esteve, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Efficient tlb-based detection of private pages in chip multiprocessors," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, Mar. 2015.

[34] D. Kim, J. Ahn, J. Kim, and J. Huh, "Subspace snooping: Filtering snoops with operating system support," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 111–122.

[35] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *43rd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2010, pp. 187–198.

[36] ——, "SCD: A scalable coherence directory with flexible sharer set encoding," in *18th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2012, pp. 129–140.

[37] M. R. Marty and M. D. Hill, "Virtual hierarchies to support server consolidation," in *34th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2007, pp. 46–56.

[38] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 2002, pp. 211–222.

[39] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[40] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[41] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[42] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2005, pp. 190–200.

[43] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 24–36.

[44] *SPEC benchmark suite release 1.0*, SPEC, Winter 1990.

[45] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.